Bluespec-7: Scheduling & Rule Composition

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology


GAA Execution model

Repeatedly:
- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic
User annotations can help in rule selection

Implementation concern: Schedule multiple rules concurrently without violating one-rule-at-a-time semantics
Rule: As a State Transformer

A rule may be decomposed into two parts \( \pi(s) \) and \( \delta(s) \) such that

\[
s_{\text{next}} = \begin{cases} 
\pi(s) & \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s \\
\delta(s) & \text{else}
\end{cases}
\]

\( \pi(s) \) is the condition (predicate) of the rule, a.k.a. the "CAN_FIRE" signal of the rule. (conjunction of explicit and implicit conditions)

\( \delta(s) \) is the "state transformation" function, i.e., computes the next-state value in terms of the current state values.

Compiling a Rule

```plaintext
rule r (f.first() > 0) ;
x <= x + 1 ;    f.deq () ;
endrule
```

\( \pi \) = enabling condition
\( \delta \) = action signals & values
Combining State Updates: strawman

\[ \pi's \text{ from the rules that update } R \]

\[ \delta's \text{ from the rules that update } R \]

\[ \pi_1 \rightarrow \text{ OR} \rightarrow \delta_{1,R} \rightarrow \text{ OR} \rightarrow \cdots \rightarrow \delta_{n,R} \rightarrow \text{ latch enable} \rightarrow R \]

Scheduler:

Priority Encoder

\[ \phi_1 \rightarrow \text{ OR} \rightarrow \delta_{1,R} \rightarrow \text{ OR} \rightarrow \cdots \rightarrow \delta_{n,R} \rightarrow \text{ latch enable} \rightarrow R \]

\[ \pi's \text{ from all the rules} \]

Scheduler ensures that at most one \( \phi_i \) is true
One-rule-at-a-time Scheduler

1. $\phi_i \Rightarrow \pi_i$
2. $\pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n$
3. One rewrite at a time
   i.e. at most one $\phi_i$ is true

Very conservative way of guaranteeing correctness

Executing Multiple Rules Per Cycle:

Conflict-free rules

Parallel execution behaves like $ra < rb = rb < ra$

Rule $a$ and Rule $b$ are conflict-free if

$\forall s. \pi_a(s) \land \pi_b(s) \Rightarrow$

1. $\pi_a(\delta_b(s)) \land \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) = \delta_b(\delta_a(s))$

Parallel Execution can also be understood in terms of a composite rule

```
rule ra (z > 10);
  x <= x + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule
```

```
rule ra_rb((z>10)&&(z>20));
  x <= x+1; y <= y+2;
endrule
```
Executing Multiple Rules Per Cycle:

*Sequentially Composable rules*

Rule \(a\) and Rule \(b\) are sequentially composable if

\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \pi_b(\delta_a(s))
\]

Parallel execution can also be understood in terms of a composite rule

\[
\text{Parallel execution behaves like } ra < rb \quad \Rightarrow \quad \text{ra_rb((z>10)&&(z>20));}
\]

\[
x \leq y+1; \quad y \leq y+2; \quad \text{endrule}
\]

**Sequentially Composable rules ...**

Parallel execution can behave either like \(ra < rb\) or \(rb < ra\) but the two behaviors are not the same

**Composite rules**

Behavior \(ra < rb\)

\[
\text{rule ra (z > 10);} \quad x \leq y + 1; \quad \text{endrule}
\]

\[
\text{rule rb (z > 20);} \quad y \leq y + 2; \quad \text{endrule}
\]

\[
\text{rule ra_rb((z>10) &&(z>20));} \quad x \leq y+1; \quad y \leq y+2; \quad \text{endrule}
\]

Behavior \(rb < ra\)

\[
\text{rule rb ra(z>10 && z>20);} \quad x \leq 2; \quad \text{endrule}
\]

\[
\text{rule rb ra(z>10 && z>20);} \quad x \leq 2; \quad \text{endrule}
\]
Compiler determines if two rules can be executed in parallel

Rule \(a\) and Rule \(b\) are conflict-free if
\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow
\begin{align*}
1. & \; \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \\
2. & \; \delta_a(\delta_b(s)) = \delta_b(\delta_a(s))
\end{align*}
\]

Rule \(a\) and Rule \(b\) are sequentially composable if
\[
\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow
\begin{align*}
1. & \; \pi_b(\delta_a(s)) \land \pi_a(\delta_b(s)) \\
2. & \; \delta_a(\delta_b(s)) = \delta_b(\delta_a(s))
\end{align*}
\]

These properties can be determined by examining the domains and ranges of the rules in a pairwise manner.

These conditions are sufficient but not necessary. Parallel execution of CF and SC rules does not increase the critical path delay.

Mutually Exclusive Rules

\(\text{Rule}_a\) and \(\text{Rule}_b\) are mutually exclusive if they can never be enabled simultaneously
\[
\forall s . \pi_a(s) \Rightarrow \lnot \pi_b(s)
\]

Mutually-exclusive rules are Conflict-free even if they write the same state

Mutual-exclusive analysis brings down the cost of conflict-free analysis
Multiple-Rules-per-Cycle Scheduler

1. \( \phi_i \Rightarrow \pi_i \)
2. \( \pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n \)
3. Multiple operations such that \( \phi_i \land \phi_j \Rightarrow R_i \) and \( R_i \) are conflict-free or sequentially composable

Muxing structure

Muxing logic requires determining for each register (action method) the rules that update it and under what conditions

Conflict Free (Mutually exclusive)

Sequently composable

CF rules either do not update the same element or are ME

\( \pi_1 \Rightarrow \neg \pi_2 \)
Scheduling and control logic

Modules (Current state)

\[ \pi_1, \delta_1 \]

\[ \ldots \]

\[ \pi_n, \delta_n \]

Rules

\[ \text{cond} \]

\[ \text{action} \]

Scheduler

\[ \phi_1 \]

\[ \ldots \]

\[ \phi_n \]

Muxing

\[ \delta_1 \]

\[ \ldots \]

\[ \delta_n \]

Modules (Next state)

cond

action

some insight

Pictorially

Rules

HW

\[ \text{rule steps} \]

\[ \text{clocks} \]

- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges
Parallel execution reorders reads and writes

- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules.
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks.

Correctness

- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution (i.e., CF or SC).
- Consequence: the HW can never reach a state unexpected in the rule semantics.
Synthesis Summary

- Bluespec generates a **combinational hardware scheduler** allowing multiple enabled rules to execute in the same clock cycle
  - The hardware makes a rule-execution decision on every clock (i.e., it is not a static schedule)
  - Among those rules that CAN_FIRE, only a subset WILL_FIRE that is consistent with a Rule order
- Since multiple rules can write to a common piece of state, the compiler introduces appropriate muxing logic
- For proper pipelining, dead-cycle elimination and value forwarding, the user needs some understanding and control of scheduling

Two-stage Pipeline

```plaintext
rule fetch_and_decode (!stallfunc(instr, bu));
    bu.enq(newIt(instr, rf));
    pc <= predIa;
endrule

rule execute (True);
    case (it) matches
        tagged EAdd(dst:.rd, src1:.va, src2:.vb): begin
            rf.upd(rd, va+vb); bu.deq(); end
        tagged EBz {cond:.cv, addr:.av}:
            if (cv == 0) then begin
                pc <= av; bu.clear(); end
            else bu.deq();
        tagged ELoad{dst:.rd, addr:.av}: begin
            rf.upd(rd, dMem.read(av)); bu.deq(); end
        tagged EStore{value:.vv, addr:.av}: begin
            dMem.write(av, vv); bu.deq(); end
    endcase
endrule
```

Can these rules fire concurrently?
Two-stage Pipeline Analysis

1. fetch < execute
2. execute < fetch

Scheduling expectations:
execute < fetch schedule

```plaintext
rule fetch_and_decode (!stallfunc(instr, bu));
    bu.enq(newIt(instr,rf));
    pc <= predIa;
endrule

rule execute (True);
    case (it) matches
        tagged EAdd{dst:.rd, src1:.va, src2:.vb}: begin
            rf.upd(rd, va+vb); bu.deq(); end
        tagged EBz {cond:.cv, addr:.av}: begin
            if (cv == 0) then begin
                pc <= av; bu.clear(); end
            else bu.deq();
        tagged ELoad{dst:.rd, addr:.av}: begin
            rf.upd(rd, dMem.read(av)); bu.deq();
        tagged EStore{value:.vv, addr:.av}: begin
            dMem.write(av, vv); bu.deq(); end
    endcase endrule
```
One Element FIFO Analysis

module mkFIFO1 (FIFO#(t));
    Reg#(t) data <- mkRegU();
    Reg#(Bool) full <- mkReg(False);
    method Action enq(t x) if (!full);
        full <= True;     data <= x;
    endmethod
    method Action deq() if (full);
        full <= False;
    endmethod
    method t first() if (full);
        return (data);
    endmethod
    method Action clear();
        full <= False;
    endmethod
endmodule

Expectation \(bu\): \((\text{first}<\text{deq}) < (\text{find}<\text{enq})\)

The good news ...

\(~\text{It is always possible to transform your design to meet desired concurrency and functionality~}\)
Register Interfaces

`read < write`  `write < read ?`

- `read'` – returns the current state when `write` is not enabled
- `read'` – returns the value being written if `write` is enabled

Ephemeral History Register (EHR)

`read^0 < write^0 < read^1 < write^1 < ....`

- `write^{i+1}` takes precedence over `write^i`
Transformation for Performance

```verilog
rule fetch_and_decode (!stallfunc(instr, bu));
    bu.enq1(newIt(instr, rf));
    pc <= predIa;
endrule

rule execute (True);
    case (it) matches
        tagged EAdd{dst:.rd, src1:.va, src2:.vb}: begin
            rf.upd0(rd, va+vb); bu.deq0(); end
        tagged EBz {cond:.cv, addr:.av}: begin
            if (cv == 0) then begin
                pc <= av; bu.clear0(); end
            else bu.deq0(); end
        tagged ELoad{dst:.rd, addr:.av}: begin
            rf.upd0(rd, dMem.read(av)); bu.deq0(); end
        tagged EStore{value:.vv, addr:.av}: begin
            dMem.write(av, vv); bu.deq0(); end
    endcase endrule
```

One Element FIFO using EHRs

```verilog
module mkFIFO1 (FIFO#(t));
    EHReg2#(t) data  <- mkEHReg2U();
    EHReg2#(Bool) full  <- mkEHReg2(False);
    method Action enq0(t x) if (!full.read0);
        full.write0 <= True; data.write0 <= x;
    endmethod
    method Action deq0() if (full.read0);
        full.write0 <= False;
    endmethod
    method t first0() if (full.read0);
        return (data.read0);
    endmethod
    method Action clear0();
        full.write0 <= False;
    endmethod
endmodule
```
### After Renaming

- Things will work
  - both rules can fire concurrently

Programmer Specifies:

$$R_{\text{execute}} < R_{\text{fetch}}$$

Compiler Derives:

$$(\text{first}^0, \text{deq}^0) < (\text{find}^1, \text{deq}^1)$$

What if the programmer wrote this?

$$R_{\text{execute}} < R_{\text{execute}} < R_{\text{fetch}} < R_{\text{fetch}}$$

---

### Experiments in scheduling

Dan Rosenband, ICCAD 2005

- What happens if the user specifies:
  
  $$Wb < Wb < Mem < Mem < Exe < Exe < Dec < Dec < IF < IF$$

  No change in rules  a superscalar processor!

**Executing 2 instructions per cycle** requires more resources but is functionally equivalent to the original design